실험 2. 간단한 입출력 실험과 타이머

1. 목적

(1) 기본적인 RTOS 기반 프로그램 수행 방식을 따라해 본다.

(2) 입출력 장치를 Access 하기 위한 API를 따라해 본다. - 지난 실험 내용 참조

(3) RTOS의 멀티쓰레드 프로그램을 배운다.

(4) 타이머 API 이용법을 익힌다.

2. 실험 배경

실시간 운영체제 (FreeRTOS)의 최소 기능인 멀티태스킹 기능과 타이머 기능을 몇 가지 API를 통하여 익힌다.

2.1 Task 선언을 위한 API

xTaskCreate() 함수(실제로는 macro)는 Task 하나를 정의한다. xTaskCreate()는 다음과 같이 정의된다.

```
portBASE_TYPE xTaskCreate(
        pdTASK_CODE pvTaskCode,              /* Task로 실행될 함수 */
        const char * const pcName,           /* 관리를 위한 Task 이름 */
        unsigned short usStackDepth,         /* 스택 크기 - 변수 개수 */
        void *pvParameters,                  /* Task에 전달할 Parameter */
        unsigned portBASE_TYPE uxPriority,   /* 우선 순위 */
        xTaskHandle *pvCreatedTask           /* 생성된 Task ID를 저장할 곳의 주소 */
);
```

다음은 위 정의에 따라 실험 2의 main()에서 주 task 하나를 선언한 예이다. Exp_2_Task() 함수가 실행될 Task로, 스택은 2048 word, 우선 순위는 IDLE task (아무것도 할 일이 없을 때, 도는 Task - 실제로는 CPU를 IDLE 상태로 만듬) 보다 1 높게 설정하고 있다.

```
xTaskCreate(Sample_Task_1,                                  /* 태스크로 실행될 함수 */
            (const signed char * const)"Sample_Task_1",     /* 이름 */
            2048,                                           /* 스택크기 */
            (void *)NULL,                                   /* parameter */
            tskIDLE_PRIORITY + 1,                           /* 우선 순위 */
            NULL);                                          /* 생성된 Task ID 저장소 */

// 다른 Task들 선언

vTaskStartScheduler();        // Never returns
```

실제 시스템에서 실행될 Task들은 모두 vTaskStartScheduler() 함수가 호출되어 스케줄러가 실행되기 전에 선언되어야 한다.

실시간 시스템 운영체제에서 우선 순위는 매우 강력하다. 철저하게 CPU는 우선 순위 기반한 Schedule을 하며, 높은 (FreeRTOS의 경우, 큰 우선 순위 값) 우선 순위 Task가 대기 상태 (timer를 기다리거나, 기타 다른 event를 기다리는 경우)에 들어갈 때만, 낮은 우선 순위의 태스크가 실행된다. 즉 우선 순위를 잘 결정해야 하며, 낮은 우선 순위 Task가 실행될 수 있도록 높은 우선 순위 Task는 자신이 할 일을 수행한 뒤, 바로 timer 대기 loop에 들어가서 대기 하여야 한다. 높은 우선 순위의 Task가 CPU를 계속 쓰면 (어떤 일을 하거나, 무한 루프를 돌거나 하면) 우선 순위가 낮은 Task들은 돌 수 없다.

우선 순위를 정하는 가장 일반적인 방법은 짧은 실행 주기를 갖는 Task에게 높은 우선 순위를 주는 것이며, 이것을 RM (Rate Monotonic) Schedule 방식이라고 한다.

2.2 FreeRTOS에서의 Timer 관련 API

타이머에 의한 대기를 하기 위한 함수는 vTaskDelay() 이며 인자는 tick 수 (FreeRTOS에서 타이머 인터럽트를 tick이라하면 그 개수)이다. 우리 실험 환경에 이식된 FreeRTOS에서 timer 인터럽트는 1초에 1000번 걸린다. (주기는 1msec), 따라서 tick 하나의 주기도 1msec 이다.

실시간 운영체제를 이식하는 환경마다 이 주기가 다를 수 있기 때문에 MSEC2TICK()과 같은 macro를 만들어 실시간을 tick 수로 바꾼다. 따라서 vTaskDelay(MSEC2TICK(50))는 호출된 시점부터 50msec를 기다린다. 즉 50msec 후에 리턴된다.

vTaskDelay()는 인자로 주어진 시간만큼을 대기하고 다시 Task가 schedule 되지만, 이와 유사한 함수로 vTaskDelayUntil()는 정해진 주기마다 실행을 할 수 있게 해준다. vTaskDelayUntil()은 기준 시간을 정하고 매 지정된 주기 시점까지 기다린다. 다음 예를 보자.

```
static
portTASK_FUNCTION(Sample_Task, pvParameters )
{
      xLastWakeTime = xTaskGetTickCount();
      while (1) {

            // Code for Sample_Task <====

            // Wait for the next cycle.
            vTaskDelayUntil( &xLastWakeTime, xPeriod );
    }
}
```

예를 들어, vTaskDelay(50)은 vTaskDelay() 함수가 호출된 시점부터 50 tick을 기다린다. 즉 어떤 작업의 실행 주기는, 작업시간 (vTaskDelay 함수에서 리턴한 후 작업시간) + 50 tick 이 된다. vTaskDelayUntil(&xLastWakeTime, 50)은 이전에 vTaskDelayUntil()에서 깨어난 (리턴된) 시점이 xLastWakeTime 변수에 기록되기 때문에 어떤 작업을 하더라도 지정한 tick 수인 50 tick 이전에 끝난다면, 이전에 깨어난 시점 후 50 msec 후에 깨어나, 정확한 주기를 지켜야하는 작업에 적합하다.

(구체적인 vTaskDelay()와 vTaskDelayUntil()에 관한 설명은 http://www.freertos.org/를 참조)

3. 동작실험

샘플 코드는 두 개의 Task를 생성하여 Sample_Task_1()은 vTaskDelay()로 500msec대기하고, 다른 태스크인 Sample_Task_2()는 vTaskDelayUntil()로 500msec를 대기하면서 각각 BARLED1, BARLED2를 깜빡 거리고, 별 의미없는 loop를 10만번 돌면서 시간을 때우는 코드이다. 실제 BARLED1, 2를 깜박거리는 동작은 닌텐도 위쪽 LCD에 그림을 그리는 동작을 포함하기 때문에 실행 시간이 필요하다.

Sample_Task_1()에서, while 문으로 구성된 task loop는 printf(), barled 변수 연산,

writeb_virtual_io() 함수 실행 시간 (즉, LED에 BARLED 그리는 시간), 그리고 vTaskDelay()에 의한 500msec 대기 시간을 모두 합친 만큼의 주기로 실행된다.

반면에 Sample_Task_2()는 vTaskDelayUntil()로 정확하게 500msec에 한번씩 깨어난다.

그 결과, vTaskDeley()를 사용하는 Sample_Task_1()의 주기는 항상 500msec 이상이 걸려 약간씩 느려지며, SampleTask2()는 언제나 같은 주기로 실행되어 BARLED2는 시간이 흘러도 언제나 1초에 한번 씩 깜빡이는 것을 확인할 수 있다.

다음은 실험의 Sample 코드인 main.c의 내용이다.

```
initDebug();
init_virtual_io(ENABLE_LED); // Enable Virtual IO Devices
init_printf();               // Initialize for printf()

static
portTASK_FUNCTION(Sample_Task_1, pvParameters)
{
  u8 barled = 0;
  int i;

  while (1) {
    printf("1");
    barled = ~barled;
    writeb_virtual_io(BARLED1, barled);

    for (i = 0; i < 100000; i++)       /* DELAY LOOP */
       barled = ~barled;

    vTaskDelay(MSEC2TICK(500));
  }
}

static
portTASK_FUNCTION(Sample_Task_2, pvParameters)
{
  u8 barled = 0;
  portTickType xLastWakeTime = xTaskGetTickCount();
  int i;

  while (1) {
    printf("2");
    barled = ~barled;
    writeb_virtual_io(BARLED2, barled);

    for (i = 0; i < 100000; i++)
       barled = ~barled;

    vTaskDelayUntil(&xLastWakeTime, MSEC2TICK(500));
  }
}
```

코드에서 InitDebug()는 Debugging을 위해서 필요한 함수로 실제 debugging을 하기 위해서는 Makefile의 35번째 줄 #CFLAGS += -DDEBUG 의 #을 제거해야한다.

init_virtual_io(ENABLE_LED)는 위쪽 LCD에 가상 디바이스인 BARLED를 초기화 하기 위한 것이고, init_printf()는 아래쪽 화면에 printf() 출력을 낼 수 있도록 하는 함수 이다.

4. 과제

과제는 main() 함수에 Sample_Task_1, Sample_Task_2를 지우고, Homework_1(), Homework_2() Task를 새로 만들어 다음과 같은 동작을 하도록 한다. Task를 초기화 할 때 우선순위는 Homework_1()이 높도록 설정한다.

Homework_1() : 프로그램이 시작되면, BARLED1의 LED 하나가 켜지고, 닌텐도 뒤 쪽의 키 KEY_L, KEY_R를 누를 때 마다, 각각 왼쪽, 오른쪽으로 한 칸씩 켜진 LED가 옮겨 가는 것이다. 켜진 LED가 BARLED1의 오른쪽 끝이면 KEY_R를 눌러도 변화가 없어야 하며, 반대로 BARLED1의 왼쪽 끝에서는 KEY_L를 눌러도 변화가 없어야 한다.
(실험 1의 과제 A와 동일, Key 만 KEY_L, KEY_R로 바뀜)

Homework_2() : 처음 BARLED2의 제일 오른쪽 LED를 켜고, 0.5초에 한번 씩 오른쪽 BARLED 하나를 왼쪽 방향으로 돌린다. 제일 왼쪽 LED가 켜진 뒤에는 다시 제일 오른쪽으로 돌아가도록 한다.

(검사 기한)

과제 A : 실험 당일

(보고서 제출)

결과 보고서 Template를 이용하여 보고서 작성 후 제출
실험 내용, 방법, 구현된 소스 설명, 시험 결과 등

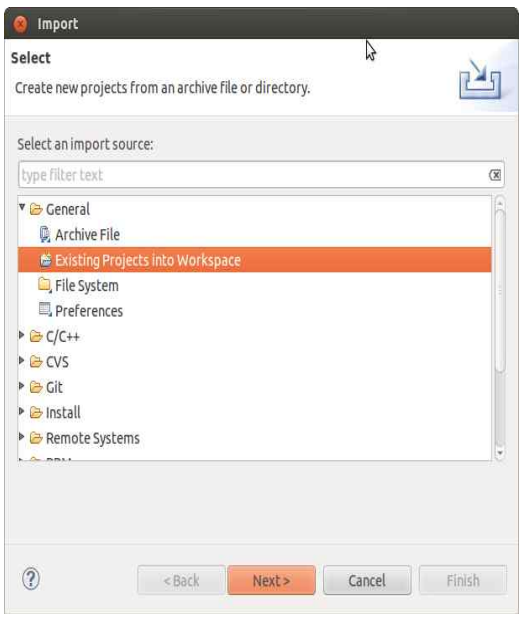
보고서에는 Homework_2() Task가 정확한 시간에 LED를 회전하는지 눈으로 관찰하고 시

계로 측정한 결과를 담는다.

4-1. 소스 및 샘플 프로그램 (강의 게시판 이용)

소스는 강의 자료실의 2-Simple-IO-2.zip를 이용하며, source 폴더의 main.c파일을 수정하여 과제를 수행한다.

소스는 workspace 폴더에 복사한 뒤, Eclipse의 [File] -> [import] 다이얼로그 박스에서 [Existing Projects into Workspace]를 선택하고 [Next->]를 누른뒤 (왼쪽 그림)
[Select root directory]를 선택하고 [Browse...]에서 /home/hansung/ndsdev/workspace (소스 zip 파일을 설치한 곳)를 지정하고 프로젝트 선택하고 [Finish]하여 적용



B. 미리 만들어 놓은 Binary

이 실험 결과가 수행되는 모습은 Sample 코드의 실행 결과는 sample-2.nds, 과제 실행 결과는 Simple-IO-2.nds를 닌텐도 DS에 다운로드하여 실행하여 확인할 수 있음

4-2. 소스 및 샘플 프로그램 (Github 이용)

소스는 Github의 저장소에서 simple-io-2 소스를 check-out하여 과제를 수행한다.

https://github.com/hl1itj/nds-ide/tree/microprocessor/lab/simple-io-2

에서, Checkout하여 과제를 수행한다.
이 문서와 미리 build한 binary인 sample-2.nds (위 샘플 프로그램), Simple-io-2.nds (과제 결과)는 doc directory에 있음

https://github.com/hl1itj/nds-ide/tree/microprocessor/학번/simple-io-2

에서 각자 개발을 진행한다.